Brennan Reid Steuwe

Oscar Contreras

Seth Thomas Giovannetti

**Server Based Chat - Team Report**

Compilation and Execution Instructions:

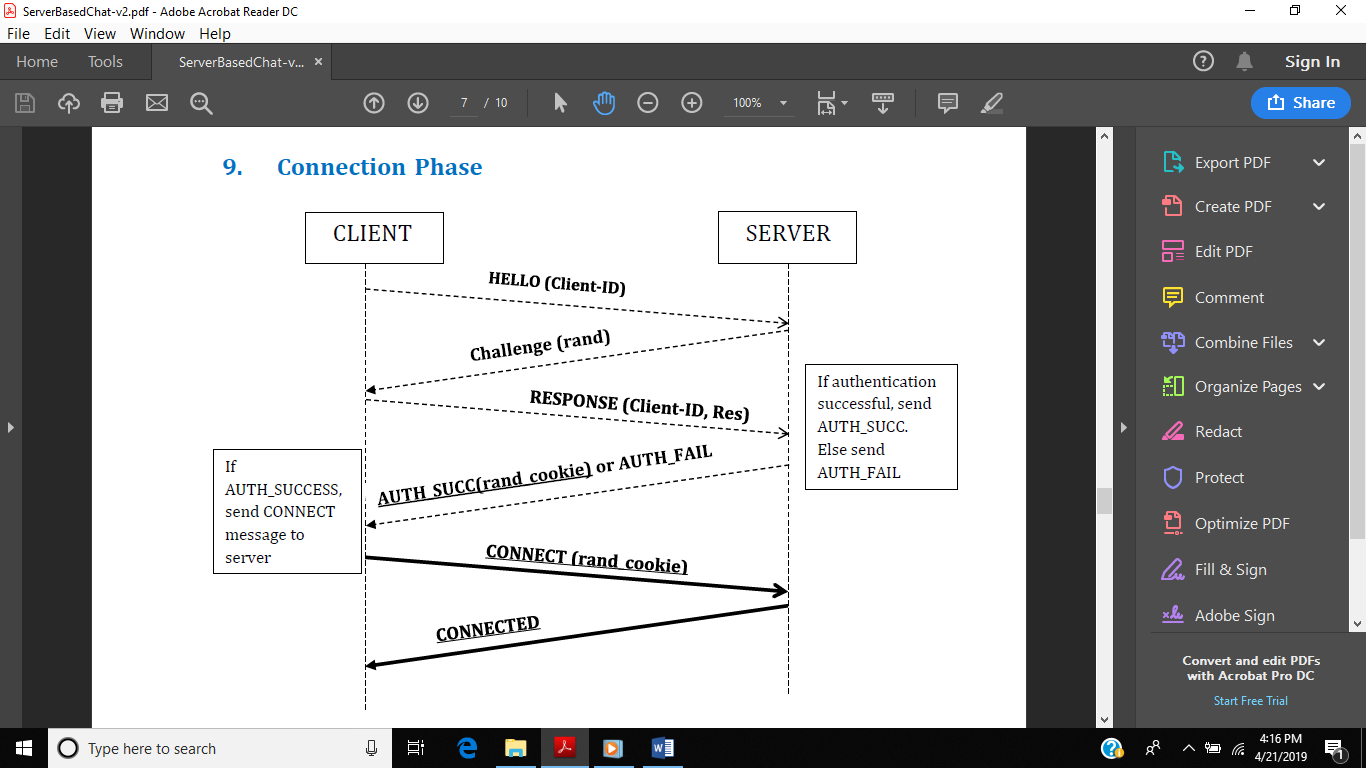
See README.md for details.

Authentication:

Although it was not required for our revised instructions, we implemented the challenge and response authentication scheme laid out in the initial project. The following sequence of steps summarizes the process needed to establish a TCP connection.

1. When a client program begins and a client\_id is entered, the client sends a HELLO message over UDP to the server.
2. On receiving the HELLO from the client, the server generates a random number and sends a CHALLENGE message over UDP if the client is a subscriber. If the client is not a subscriber, the server sends a DECLINED message over UDP.
3. On receiving the CHALLENGE from the server, the client does the following:
   1. If needed, generates a secret ciphering key and stores this in a folder named ‘keys’ (the folder will contain key files with the notation client\_id.key).
   2. Runs an A3 authentication algorithm (SHA256) with the random number sent by the server and the client\_id as inputs.
   3. Sends a RESPONSE message over UDP to the server, containing the result RES of the A3 algorithm.
4. On receiving the RESPONSE from the client, the server:
   1. Runs the same A3 authentication algorithm using the random number from step 2 and the stored client key.
   2. Compares its result XRES with the result given by the client.
      1. If these are not equal, the server sends an AUTH\_FAIL message over UDP to the client and an error is raised.
      2. If there are equal, the server sends an AUTH\_SUCC message over UDP to the client with a token (rand\_cookie, another generated random number) and TCP port to establish a connection.
5. On receiving an AUTH\_SUCC message from the server, the client opens a TCP socket and sends to the server a CONNECT message over TCP containing the token.
6. On receiving the CONNECT message, the server verifies the token and sends a CONNECTED message to the client over TCP.

Sequence Diagram of Connection Establishment:



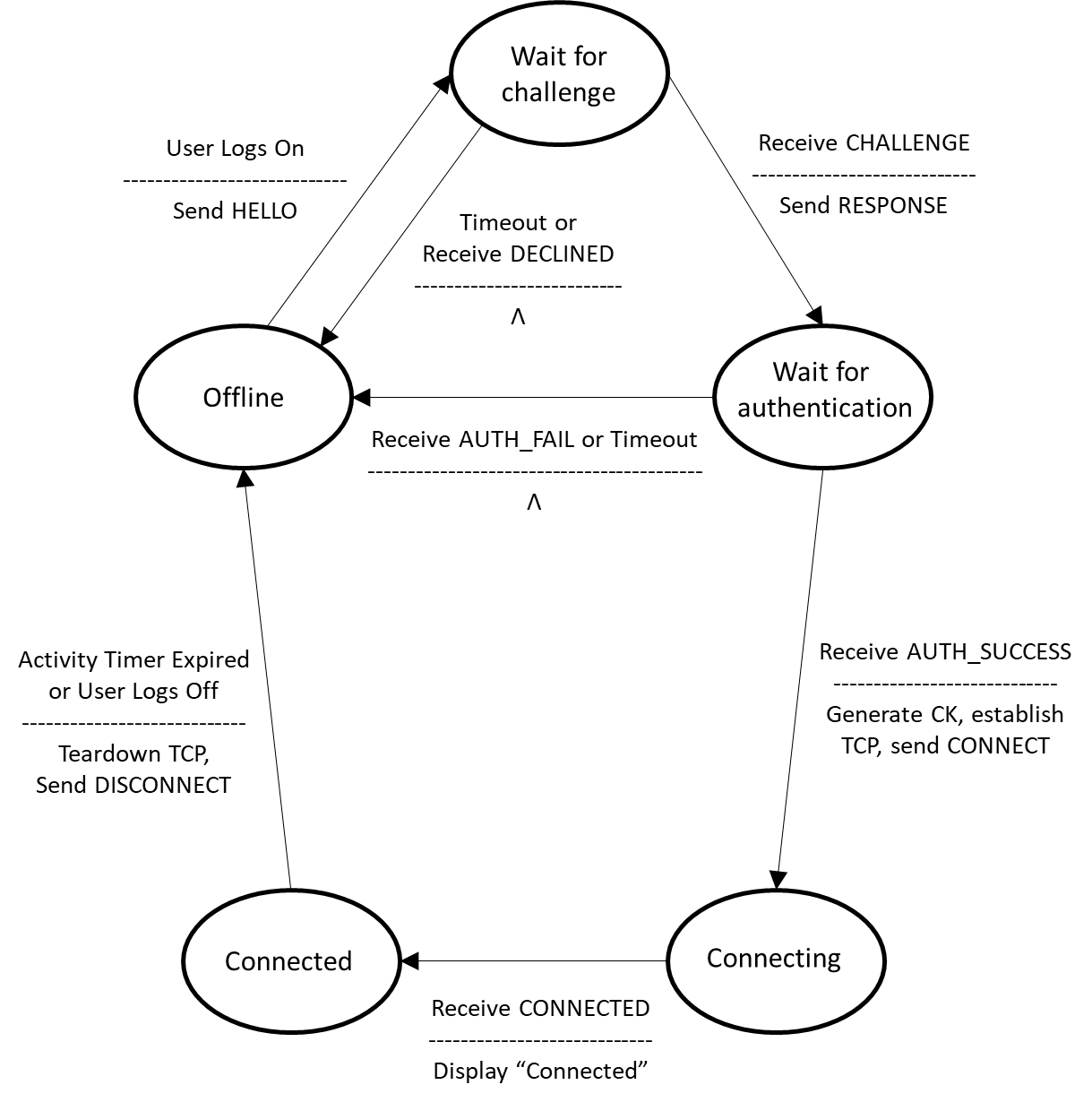
Hardware Setup and Configuration:

OS: Windows 10

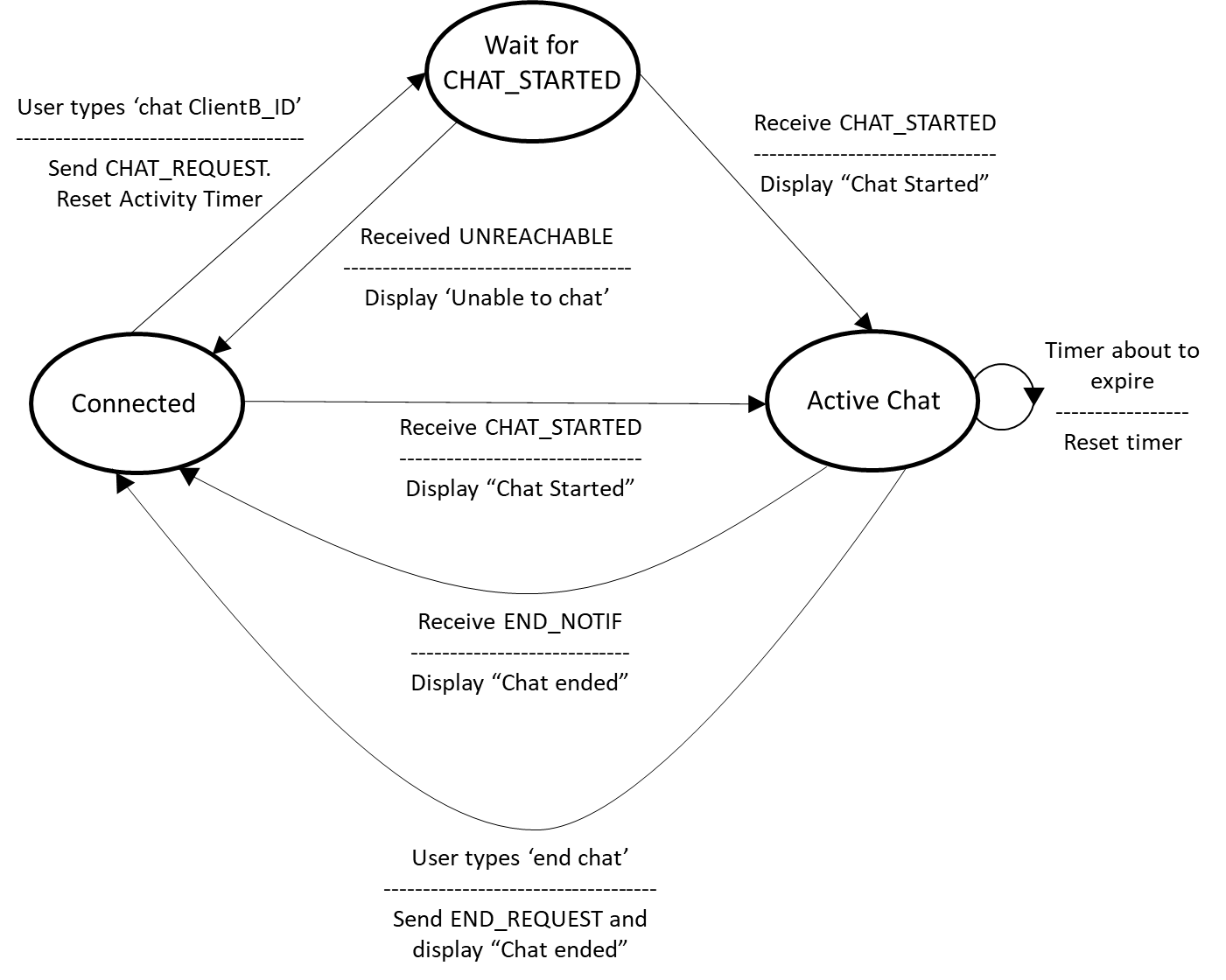
Python Version: 3.7

Required Modules: prompt\_toolkit

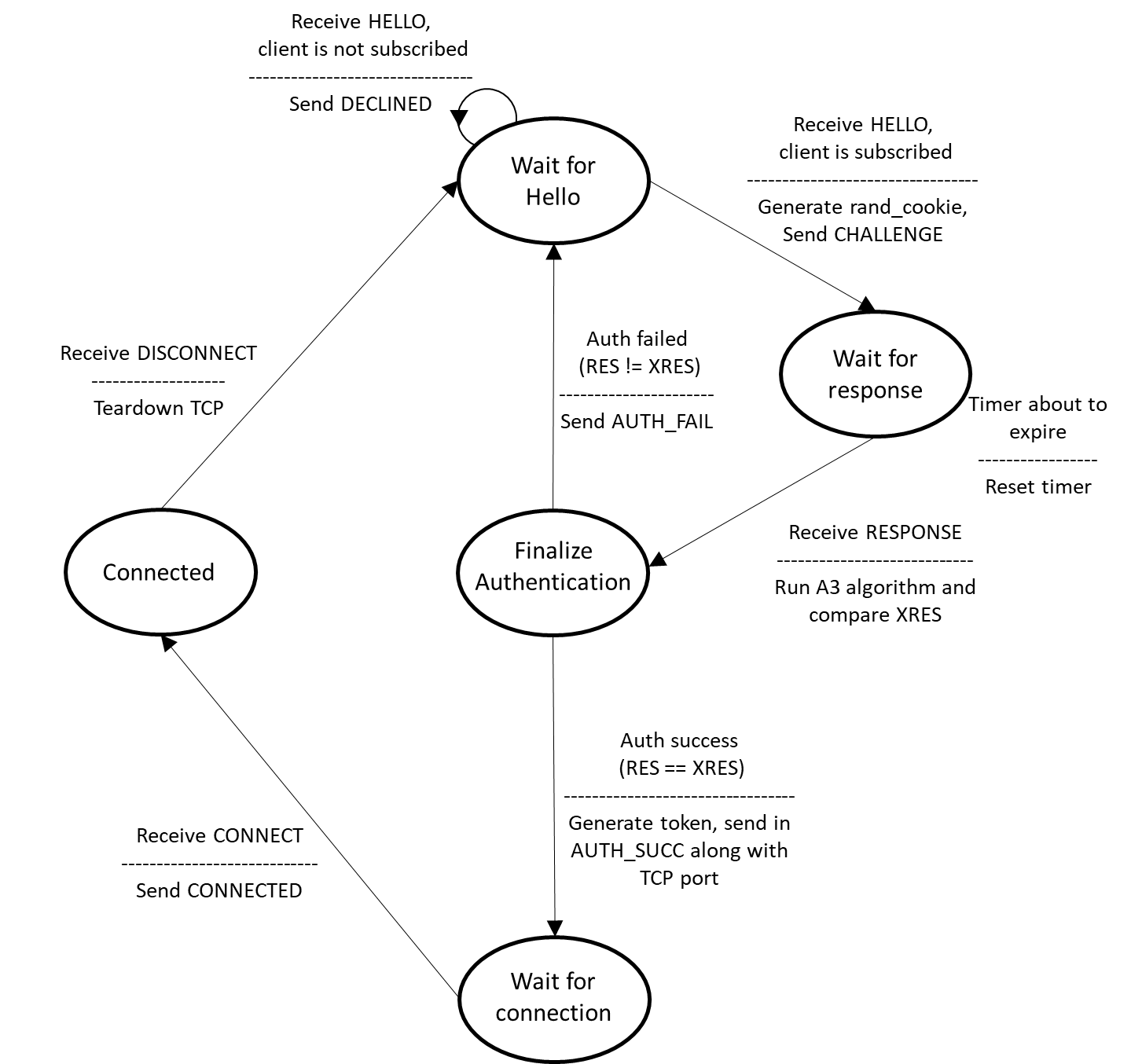
Design Documents:

**Client - Connection Phase:**

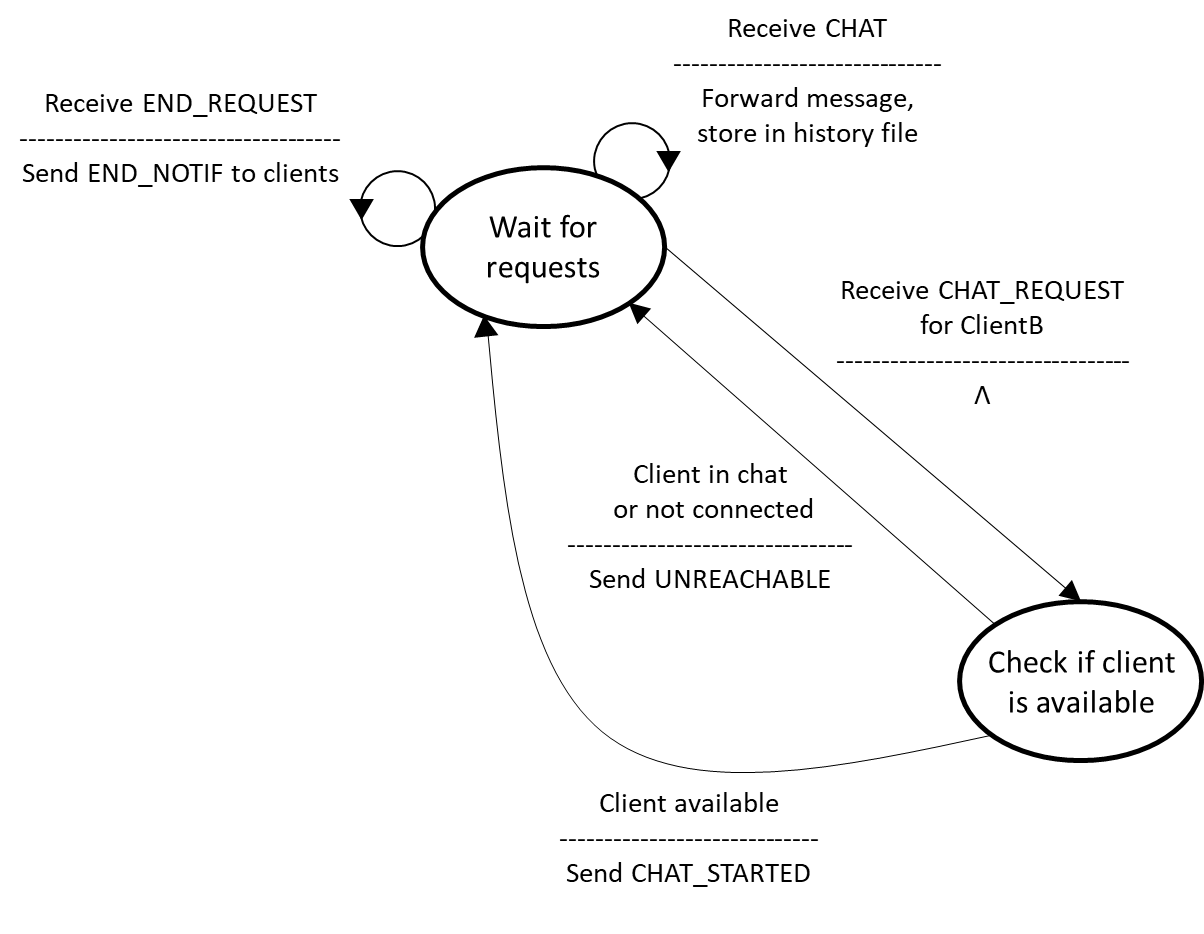
**Client - Chat Phase:**

****

**Server - Connection Phase:**

****

**Server - Chat Phase:**



Protocol Messages:

* HELLO (Client-ID-A): Initiates the process for Client A to be authenticated and registered with the server.
* DECLINED: Sent by the server if a client is not a subscriber to the chat.
* CHALLENGE (rand): Sent by the server to challenge the client to authenticate itself. rand is a random number generated by the server. A new rand is generated at every challenge.
* RESPONSE (client-ID, Res): Response to the challenge, sent by the client to authenticate itself.
* AUTH\_SUCCESS (rand\_cookie, port\_number): Sent by the server to notify the client authentication is successful. rand\_cookie is a random number generated by the server, and port\_number is a TCP port number assigned by the server for subsequent connection by the client.
* AUTH\_FAIL: Sent by the server to notify the client authentication has failed.
* CONNECT (rand\_cookie): Sent by the client to the server. rand\_cookie is the cookie previously sent by the server.
* CONNECTED: Sent by the server to notify the client it has been connected.
* CHAT\_REQUEST (Client-ID-B): Sent by client A to the server to request a chat session with Client B.
* CHAT\_STARTED (session-ID, Client-ID-B): Sent by the server to notify client A that a chat session with client B has started. Session-ID is an ID assigned to the session.
* UNREACHABLE(Client-ID-B): Sent by the server to client A to notify that the client B is not available for chat.
* END\_REQUEST: Sent by any of the clients involved in the chat session to request a termination of their current session.
* END\_NOTIF (session-ID): Sent by the server to notify a client involved in the session that the session has been terminated by another client.
* CHAT (chat message): Exchanged between the clients, relayed by the server. Carries the actual chat message.
* HISTORY\_REQ (Client-ID-B): Sent by client A to request the history of past chat messages with client B.
* HISTORY\_RESP (Sending Client-ID, chat message): Sent by the server to the client who requested the history. Sending Client-ID is the ID of the client who sent the chat message, and chat message is the chat message in the history. There is one HISTORY\_RESP message for each chat message in the history.

Screenshots:

TODO - show main validation scenarios

Issues during development:

One of the most significant issues we faced in the project was finding a way to separate history response messages on the client side, as TCP is a byte-stream oriented protocol. When such messages were sent over TCP, they would essentially be massed together on the client end leading to garbled output and internal errors related to unexpected argument lengths. To solve this and keep track of the separate history response messages on the client end, we added our own message separator byte as a delimiter. This allows the TCP listener on a client to correctly parse the stream it receives and display the chat history as it was sent from the server.

Another byte-related issue we faced had to do with our implementation of the codes for each message. Each of the message types was assigned a byte code (e.g. b`\x01` for the HELLO message). For a time, we had the same byte for our message delimiter and RESPONSE messages for client authentication, leading to an ambiguous error stemming from how the bytes were parsed. We solved this by ensuring that each of these important codes or values had a unique byte to identify it, preventing such ambiguous conflicts.

Finally, a small but interesting issue we found was related to our implementation of TCP listeners on the server using threads. These would essentially run a routine in an infinite loop, waiting for incoming bytes from a client TCP socket. When disconnecting a client, we would find that an error was being thrown on the server. However, no functionality would actually be disrupted. The underlying reason was that when the client was no longer connected, the TCP listener would attempt to listen on a now-closed socket. This was fixed by catching the error and properly terminating the listener thread.

TODO: Video Demo